Project 4 Report

Arteen Abrishami

University ID 205-577-156

CS 32

Ambrosio

1. Function Descriptions

Class Hierarchy

Actor

Human

TunnelMan

Protester

RegularP

HardcoreP

Earth

Boulder

Squirt

Barrel

Nugget

Sonar

Water

StudentWorld

class StudentWorld : public GameWorld

{

public:

StudentWorld(std::string assetDir)

: GameWorld(assetDir)

{

}

// can't place another constructor b/c it wouldn't initialize gameworld(assetdir)

virtual ~StudentWorld() {}; // made virtual, alongside all other constructors, for polymorphism/inheritance purposes

Earth\* getEarths(int x, int y) const {return earths[x][y];} // returns the earth at the inputted position, allows access for any of the other classes into the earths array to check whether earth exists there or not

void earthsKiller(int x, int y); // destroys the earth at the given position (sets to invisible) and assists in playing the digging sound noise

void generateBoulderValues(int& x, int& y); // generates values for boulder placement

void deleteBoulderEarth(int x, int y); // deletes the Earth behind the boulder placed

bool checkBoulderBlocking(GraphObject::Direction d, int x, int y); // checks whether there is a boulder blocking in that direction (extremely convoluted calculations regarding distance performed) - one of the hardest functions to implement

void runawayBoulder(int x, int y); // performs boulder's action of falling and killing anything within its distance

bool checkEarthBlocking(GraphObject::Direction d, int x, int y); // performs a check to see whether or not there is earth blocking movement in that direction

bool killerSquirt(int x, int y); // performs the squirt's kill action (killing actors)

void squirtGun(GraphObject::Direction d, int x, int y); // creates and animates the squirt in the oil field

double distanceToTunnelMan(int x, int y); // how far r u from tunnelman? useful for gold and barrels, and protesters

bool closeToProtester(int x, int y); // allows pickup for gold, closely related to the above, for bribing purposes

void dropGold(int x, int y); // called by tunnelman-TAB

void increaseGold(); // tells tunnelman that he has picked up gold

void increaseSonar(); // again, for the tunnelman, you've picked up sonar

void illuminate(int x, int y); // illuminates the distance specified in the spec of the radius from tunnelman, called by pressing character 'z' or 'Z'

void increaseWater(); // for tunnelman

bool isClear(int x, int y); // for adding water, checks to see whether or not the space there is an allowable space for the addition

bool checkFacing(GraphObject::Direction, int, int); // checks whether or not the protester is currently facing the tunnelman

void annoyTunnelMan(); // annoys the tunnelman (damage and so on - assists in communication between classes)

int tunnelmanGetX(); // returns tunnelman's X (for protester)

int tunnelmanGetY(); // implemented in .cpp because class forwarding

virtual int init(); // spec related

virtual int move(); // spec related

virtual void cleanUp(); // spec related

void decreaseBarrels() {m\_barrels--;} // self explanatory

void generateBoldValues(int& x, int& y); // barrels and gold, generates values, for placement

enum State {falling, waiting, stable, alive, dead, pickupableP, pickupableT, stunned}; // stunned is specifically so can't hit protesters who are stunned

// allows to set the state for each actor, each one uses specific ones while others don't

struct Point

{

public:

Point(int x, int y) : m\_x(x), m\_y(y) {}

Point(){}

int getX() const {return m\_x;}

int getY() const {return m\_y;}

void setX(int x) { m\_x = x;}

void setY(int y) { m\_y = y;}

private:

int m\_x;

int m\_y;

}; // for shortest path, like in lecture #5, allows going backward through it and passing movement by the x and y instead of direction (for the protester tracking)

class Actor : public GraphObject

{

public:

// constructor

Actor(int imageID, int startX, int startY, Direction startDirection, double size, unsigned int depth)

: GraphObject(imageID, startX, startY, startDirection, size, depth) { setVisible(true); } // for Earth only

Actor(StudentWorld\* world, int imageID, int startX, int startY, Direction startDirection, double size, unsigned int depth)

: GraphObject(imageID, startX, startY, startDirection, size, depth)

, m\_world(world) {setVisible(true);} // visible unless I tell u otherwise

virtual ~Actor(){}

virtual bool blocksOthers() {return false;} // true for boulder

virtual void decreaseHealth(int howMuch) {} // upcasting purposes (so that can be called from vector of actors)

virtual bool annoyable(){return false;} // true for humans, differentiation purposes

virtual void pickedUpGold() {} // upcast for protesters, only accessable if annoyable

virtual void doSomething() = 0; // must instantiate this class (everything does something - Earth is special empty case)

void move(Direction); // moves it in given direction

void moveOne(int x, int y); // specifically for protesters, cleans stuff up, takes care of direction setting, moves them toward their proper path

// originally named moveTo until I became aware of my retardation

bool checkAnyBlock(Direction, int, int); // given any direction and a coordinate, check whether there is some sort of block (earth, boundary, or boulder)

void beginCountdown(int c) { count = c; } // used by all classes (or at least most) for very special purposes (very useful)

void decrementCount() {count--;}

bool itsTime() {if (count <= 0) return true; else return false;}

State getState() const {return s;} // same as above

void setState(State input) {s = input;}

StudentWorld\* getWorld() { return m\_world; }

private:

StudentWorld\* m\_world;

State s;

int count;

};

class Earth : public Actor

{

public:

Earth(int x, int y) : Actor(TID\_EARTH, x, y, right, 0.25, 3) {}

virtual ~Earth(){}

virtual void doSomething() override {} // override

};

class Boulder : public Actor

{

public:

Boulder (StudentWorld \*world, int x, int y) : Actor(world, TID\_BOULDER, x, y, down, 1.0, 1) {setState(stable);}

virtual ~Boulder() {}

virtual void doSomething() override; // does all according to spec

virtual bool blocksOthers() override {return true;} // for diferentiation

bool mustStop();

};

class Squirt : public Actor

{

public:

Squirt(Direction d, int x, int y, StudentWorld \*world) : Actor(world, TID\_WATER\_SPURT, x, y, d, 1.0, 1)

{setState(falling); beginCountdown(4);} // instantly falling, use countDown method

virtual ~Squirt(){}

virtual void doSomething() override; // according to spec

};

class Barrel : public Actor

{

public:

Barrel(StudentWorld\* world, int x, int y) : Actor(world, TID\_BARREL, x , y , right, 1.0, 2)

{setVisible(false); setState(pickupableT);}

virtual ~Barrel(){}

virtual void doSomething() override; // spec

};

class Gold : public Actor

{

public:

Gold(State s, StudentWorld\* world, int x, int y) : Actor(world, TID\_GOLD, x, y, right, 1.0, 2)

{ setState(s);

if (s == pickupableT) setVisible(false);

else

beginCountdown(100); // lasts 100 ticks once TAB pressed

} // either starts out pickupableT or pickupableP and that determines visibility, so we could pass its state to it // becomes visible and pickupableP when dropped by tunnelman

~Gold(){}

virtual void doSomething() override; // spec

};

class Sonar : public

{

public:

Sonar(StudentWorld\* world, int x, int y);

virtual ~Sonar(){}

virtual void doSomething() override; // spec

};

class Water : public Actor

{

public:

Water(StudentWorld\* world, int x, int y);

virtual ~Water(){}

virtual void doSomething() override; // spec

};

class Human : public Actor

{

public:

Human(int health, StudentWorld\* world, int imageID, int startX, int startY, Direction startDirection, double size, unsigned int depth)

: Actor(world, imageID, startX, startY, startDirection, size, depth) , m\_health(health)

{setState(alive);}

virtual ~Human(){}

virtual bool annoyable() override {return true;} // differentiation

virtual void decreaseHealth(int howMuch) override { m\_health -= howMuch;} // all humans have health

int getHealth() const {return m\_health;} // for tunnelman to declare deadness, and also protesters

private:

int m\_health;

};

class Protester : public Human // a base class

{

public:

Protester(int health, StudentWorld\* world, int imageID); // moved to cpp for cleanliness

virtual ~Protester(){} // no dynamic allocation, only objects of the type and then pointers to those objects that are objects of Protester

bool staticBehavior(); // same for both hardcore and regular protester - resting, falling, dead, or shouting covered in this, half of doSomething, if this happens, then none of the other behaviors happen, hence true/false

void movementBehavior(); // other half of shared behavior, mostly regarding movement while alive (in between goes hardcore protester's tracking abilities)

// ABC because no doSomething()

bool checkStraightPath(int x, int y, GraphObject::Direction d); // could I move the whole way in the direction given to me and get to where tunnelman is without any sort of blockage?

void shout(); // force interaction with tunnelman

bool findShortestRoute(int endx, int endy, std::stack<Point\*> & S); // uses BFS to get to x, y (60 for the exit route, may be different when involves tunnelman - > variable), true if solvable, false if not

virtual void decreaseHealth(int howMuch) override; // for annoyance

virtual void pickedUpGold() override {} // simply as a reminder to myself that this must be overriden in the following classes (and to make sure nothing happened if i didn't)

protected:

int restingTicks; // for adding gold

int ticksSinceLastShout; // for hardcore to increment

private:

int numSquaresToMoveInCurrentDirection;

int ticksToWaitBetweenMoves;

std::stack<Point\*> S; // holds points for exit route

bool visited[64][64]; // u been here before?

Point points[64][64]; // can only go from 0 - 60, the rest are for Earth, checkAnyBlock takes care of that

Point\* previous[64][64]; // not

};

class RegularP : public Protester

{

public:

RegularP(StudentWorld\* world) : Protester(5, world, TID\_PROTESTER) {}

virtual ~RegularP(){}

virtual void doSomething() override; // per the spec

virtual void pickedUpGold() override; // lets the protester know that gold has been picked up and performs actions related to it

virtual void decreaseHealth(int howMuch) override; // performs actions related to protester annoyance by squirt

};

// int ticksToWaitBetweenMoves = max(0, 3 – current\_level\_number/4)

class HardcoreP : public Protester

{

public:

HardcoreP(StudentWorld\* world) : Protester(20, world, TID\_HARD\_CORE\_PROTESTER) { M = 16 + getWorld()->getLevel() \* 2;}

virtual ~HardcoreP(){}

virtual void doSomething() override; // used static and movement behavior and added the one extra in between #5 in spec, specifically relating to tracking behavior

virtual void pickedUpGold() override; // for hardcore protester's bribe functionality

virtual void decreaseHealth(int howMuch) override; // for annoyance functionality

private:

std::stack<Point\*> T; // for tunnelman

double M; // double allows comparison to distanceToTunnelMan

};

class TunnelMan : public Human

{

public:

TunnelMan(StudentWorld \*world) : Human (10, world, TID\_PLAYER, 30, 60, right, 1.0, 0) {m\_water = 5; m\_sonar = 1; m\_gold = 0; beginCountdown(4);}

virtual ~TunnelMan(){}

virtual void doSomething() override; // does as per the spec

void dig(); // performs digging action and communicates with studentworld

void increaseGold() {m\_gold++;}

void increaseSonar() {m\_sonar++;}

void increaseWater() {m\_water++;}

int getGold() const {return m\_gold;}

int getWater() const {return m\_water;}

int getSonar() const {return m\_sonar;}

private:

int m\_water;

int m\_sonar;

int m\_gold;

};

**2. Unimplemented Functionality/Known Bugs**

All functionality was finished, and there are no known bugs.

**3. Design Decisions/Assumptions**

None. Everything was according to spec.

**4. Class Testing**

In addition to individual class testing, I played the game up to around level 20 myself, and everything was working perfectly. I also played it at random intervals just for fun, and I couldn't ask for it to be working any better. I am writing the reports below based on order of implementation.

Earth:

I simply made sure that the array was set up correctly and the field populated correctly. I set up the earth array to assist in other functions to do with interactions with the other classes, and everything seemed to go smoothly with it. There are no issues with earth whatsoever, and I am not sure what else I can write on it.

TunnelMan:

Firstly, I made sure that all the stuff to do with his image and contents was perfectly fine, and I checked this using the status bar at the top. Then, I made sure that his doSomething was working properly. I travelled from edge to edge and discovered some issues, so I set up boundary checks. I had issues with deleting earth, so I used visibility instead, and I fixed the dig sound to be at an interval of once every 4 earth blocks destroyed/set invisible so as to avoid lag/crackling noises created from playing the dig sound too much. I also made sure that every single key was performing its functionality/playing the appropriate sound correctly. Otherwise, there were no issues with tunnelman, and I checked his relations to do with the other classes, most predominantly boulders and protesters, later on in my testing as there wasn't much more to test at that point.

Boulder:

This was something that I spent some time on. Firstly, I made sure they were populating the field correctly at the correct locations, and this is mostly reflected in my generateBoulderValues() function. Then, I made sure that they were blocking properly; they weren't. This led to a 4-6 hour hiatus figuring out how to make them block properly and is something I am very proud of. Since the distance of 4 from the. left corner wasn't enough as at a maximum the distance would be 4 times root 2, I spent some time performing individual calculations from the points between the 1 x 1 squares occupied by the image, and ended up perfecting this in my checkBoulderBlocking() function in studentworld that checks whether or not there would be an overlap from any given direction with specific calculations from that direction, taking into account up and down movement.

Afterward, I made sure that it was falling, dying, and killing things properly, which it was, so there was no more to test (and I kept an eye on it as I moved on as well).

Squirt:

I implemented the functionality to do with damaging protesters well in advance because I already had an idea of how I wanted to do it. I tested to make sure that it was appearing (and not appearing) properly based on the amount of water the tunnelman had and also based on how close a tunnelman was to a block (as it wouldn't show up if he were closer than 4 to a block), and this includes protesters, earth, boulders, and boundaries, and I made sure the sound effects and the travel were working otherwise, which they were. Overall, it was incredibly simple to implement and observe.

Barrel:

Another very simple one, I simply made sure they were spawning correctly, appearing correctly (whether by radius to tunnelman, or later on, by sonar), and I made sure they could be picked up, the sound effect played, and the barrels decremented so as the levels could be continued.

Gold:

Another very simple one, with the added nuance of a countdown related to the protesters begun upon its second reincarnation after being dropped by a tunnelman (in which the state is different than it would be originally). I again made sure that the appearing and such was occuring correctly, and that the interactions with the tunnelman were correct. Later on, I made sure that the interactions with the protesters were correct, which they were, and a lot of the more difficult functionality for that was taken care of by the respective protesters. I also made sure it was dissapearing correctly after the appropriate ticks lapped after being dropped by tunnelman.

Sonar:

Again, similar testing process to gold and barrel. Made sure it spawning correctly and also activating correctly. I made sure it was incrementing correctly on the status bar, ,and made sure that it could be used correctly and would illuminate the nearby objects. I also made sure it was dissapearing correctly.

Water:

Again, made sure it wasn't staying around forever and dissapearing correctly, incrementing tunnelman's count correctly, and otherwise interacting correctly with tunnelman and its environment (not spawning in the wrong places).

Protester:

This one was tricky and led to a lot of debugging, specifically to do with my findShortestPath function. Since I had gone on a 12 hour coding extravaganza, what was funny is I didn't make a lot of mistakes with the actual logic of my code, but I ended up superimposing things like x and y and getX() and getY() with one another, which led to a lot of fun debugging and weird functionality. This one was also fun to test the behavior of, and actually took a substantial amount of observation, all of which I probably won't note, but will attempt to:

1. I made sure that it would come toward the tunnelman when it was in a straight path

2. I made sure it wouldn't go outside its boundaries / blocks allowed it

3. I made sure it would stand by tunnelman and shout

4. I made sure it would be making its perpendicular turns

5. I made sure it would randomly change its direction when it was supposed to

6. I made sure that it could follow mazes and stuff to a decent basis (based on the perpendicular turn)

7. I made sure it was bribed correctly

8. I made sure it could be killed correctly

9. I also made sure that it would find its path back correctly

10. I made sure the boulders could also kill it completely

There are definitely other things I made sure that I'm forgetting, but those are the ones that come to mind.

Hardcore Protester:

Similar to the regular protester, but I made sure that it could track the tunnelman based on the variable M. It would originally be 16 (which is a much shorter distance than I'd thought), and change level to level. This was pretty simple to implement after figuring out and expanding upon the findShortestPath (since it was already using endpoints, this was easy to do, but simply required another stack), and I also made sure that their bribed behavior were different and they were nigh on impossible to kill with squirts. They are very good at tracking the tunnelman in later levels when their radius is heavily increased, and it's pretty cool to see them constantly readjusting and tracking down the different mazes I made to find the tunnelman.

And that's all. Probably 60 or so hours put in, but all of it worth it. I feel like I learned a lot, and my brain is fried. Will have to take many days off my computer once the class is over to make this migraine go away.